THE ARCHITECTURE OF DDM1:
A RECURSIVELY STRUCTURED DATA DRIVEN MACHINE

by

Professor A. L. Davis

UUCS - 77 - 113

TO APPEAR IN PROCEEDINGS OF THE
FIFTH ANNUAL SYMPOSIUM ON
COMPUTER ARCHITECTURE
IN APRIL 1978

October 7, 1977

The Architecture of DDM1: A Recursively Structured Data Driven Machine

Professor A. L. Davis
Computer Science Department
University of Utah
Salt Lake City, Utah 84112

ABSTRACT

An architecture for a highly modular, recursively structured class of machines is presented. DDM1 is an instance of such a machine structure, and is capable of executing machine language programs which are data driven (data flow) nets. These nets may represent arbitrary amounts of concurrency as well as arbitrary amounts of pipelining. DDM1 is a fully distributed multi-processing system composed of completely asynchronous modules. The architecture allows for limitless physical extensibility without necessitating special programming or special hardware to support individual machines of widely varying sizes. DDM1 is capable of automatically and dynamically allocating concurrent tasks to the available physical resources. The essential characteristics of the highly parallel, pipelined machine language are also described along with its method for execution on DDM1.

I. Introduction

DDM1* (Data Driven Machine #1) was built in an attempt to investigate some ideas about machine organization to support very high levels of concurrency. The machine language programs of DDM1 are called Data Driven Nets or DDN's. DDN's were originally described in the author's thesis [1], and later refined in [2], and [3]. DDN's are similar to the net representations developed by Dennis [4] and Rodriguez [5]. The main advantages of DDN's over these other net representations are in the lack of distinction between control and data tokens, and a more general selection of primitive elements. The result is increased simplicity and clarity of the net programs. A linearized encoding of DDN's is the actual machine language of DDM1. The highly parallel, distributed, and asynchronous nature of DDN's implies that an efficient executing engine for these nets should possess similar characteristics. Other goals for a data driven machine are that it be efficient and economical with respect to current technology.

A recursive architecture was chosen to meet the above criteria. In defining a new set of guidelines for recursive machines and computing technology, Glushkov, et. al [6] proposed with uncanny accuracy, for these purposes, the following principles:

P1. Recursive machines contain a limitless number of levels of machine language.

---

*Note: DDM1 is a module of the architecture described here, and was built while the author was at Burroughs ASDO. DDM1 was completed in July 1976 and now resides at the University of Utah where the project is continuing under Burroughs support.

P2. All program elements for which operands are available are to be executed.

P3. Memory structure should be reprogrammable in order to convert the structure of data and programs as required by internal level changes.

P4. There should be no limit to the number of allowed machine elements.

P5. These machines should have a flexible, reprogrammable structure.

P1 and P2 are direct fits with the choice of a recursive architecture for a data driven machine. By recursively structured, it is meant that at any level of the architecture the structure at that level is the same as the structure at any other level. In a binary tree for example the structure at any node is that there is the possibility for a single parent node and two child nodes.

Recursive architectures actually imply P4 and the ability of the machine structure to be extensible. A modular extensible structure is extremely attractive in a cost sense for todays LSI component technology. Each new LSI module is very expensive to create but reproductions of these modules are very cheap. By minimizing the number of module types and allowing these types to be used repetitively in an extensible recursive structure, the resulting cost structure becomes very attractive. Furthermore the ability of such a machine structure to fulfill the needs for machines of all sizes, implies that support efforts may be limited to a single class of machines. The obvious goal is to do this in a way that machine performance increases at a reasonable rate as the size and cost are increased. For DDM1's architecture there is no physical limit to the number of modules or the number of recursively organized levels that may exist in a particular machine.

A number of parallel machine projects operating under sequential control have generated considerable doubt that centrally controlled parallel machines can be efficient. Such systems suffer the further disadvantage in that they cannot be arbitrarily extended without running into electrical problems such as clock skew. These problems can only be overcome by special tuning of the circuits for larger structures, which in turn decreases the efficiency attainable by an individual module. This situation can be resolved by a new principle.

P6. Modules of recursively structured machines should function in a fully distributed asynchronous manner.

Fully distributed systems are defined here to have two principal characteristics:

C1. At no time can a module of a fully distributed system determine the total system state.

C2. A fully distributed system is incapable of enforcing simultaneity in its distributed modules.

C1 does not imply that a module functions with no information about what is happening in other parts of the system. It does mean that a module cannot rely on another part of the system being in a certain configuration at any particular time. C2 augments this idea. For distributed systems, simultaneity cannot physically be enforced (except in trivial cases where for example it might be said that some boolean variable V becomes false simultaneously with $\overline{V}$ becoming true). This results in the necessity for modules to function only on the basis of the module's own local time, hence the term self-timed or asynchronous modules. The asynchronous protocol chosen for the self-timed modules of DDM1 is the standard four-cycle

request acknowledge scheme. The benefit of these self-timed modules and the recursive structure of the machine is that modules may be added without limit and no special hardware modules, tuning, or programs need be added to support these arbitrarily larger structures.

Another major goal of DDM1 is to place many of the current operating system functions into the hardware in order to increase system efficiency. Most parallel machine projects to date have required either that programs be written to take advantage of a particular hardware configuration, or that a configuration dependent compile operation be done prior to execution. Resource allocation is done dynamically and automatically by the DDM1 hardware and depends on the available concurrency in the executing process and the availability of physical resources.

The architecture of DDM1 nicely fits the stated principles. It fits P5 in that the architecture is flexible in a logical sense but not in a physical sense as in the machines advocated by Miller [7]. DDM1's architecture is a fully distributed, asynchronous, recursive structure capable of containing an unlimited number of modules. The number of actual module types however is limited to five. Machines of this architecture are felt to be more efficient and cost effective than other data flow architectures, [8] and [9] respectively.

II. Characteristics of the Machine Language

Only the essential characteristics of the DDN machine language will be presented here. Details of the language can be found in [3].

DDN's are cyclic, bipartite graph structures consisting of a collection of cells and a collection of directed data paths interconnecting these cells. Quantum units of information called data items travel over the data paths from one cell to another. Data items are typed and may be anything such as program, message, vector, scalar, etc., and are variable in length. Data paths function as FIFO storage devices when more than one data item is present. The lengths of these queues are constrained operationally within DDM1 by the availability of physical resources. It will suffice to consider data paths as queues of finite but indefinite length. There are several types of DDN cells in the actual machine language. These cell types allow such program constructs as calls (open, closed, and recursive), conditionals, arbitration, iteration, distribution of data items, deterministic merging, and synchronization. The semantics associated with cell types will be abstracted here by simply assigning a cell function or name (describing the cell's function) to each individual cell.

When a certain set of a cell's input data paths (called the firing set) contain at least one item, that cell is said to be fireable. A cell fires at some finite, but unspecified time after it becomes fireable. This notion of a finite, but unspecified delay is essential if the schema is to fit within the functional framework of a fully distributed asynchronous environment. When a cell fires the set of items comprising the firing set (the head items on the firing set data paths) are destroyed, and a set of resultant data items are placed on some set of output data

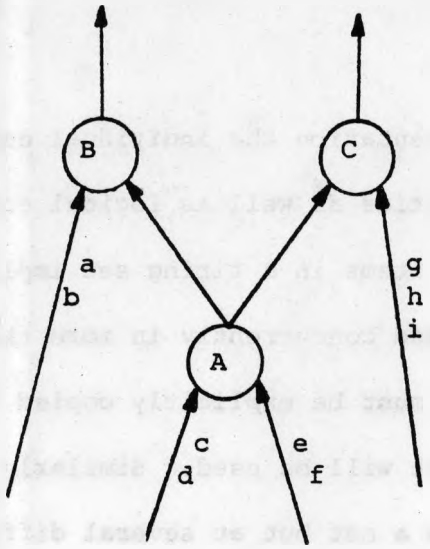
paths. The values of these resultant items and the selection of data paths on which they are placed, depend upon the cell function. No assumption is made about the relation between the times at which the items appear on the output data paths or the order in which they appear. A cell is said to have fired only after all of the firing set data items have been destroyed and all output items have been placed on the desired output paths.

There are several methods of implementing the management of data items under the above firing strategy using pointers and other forms of shared storage. All these common storage methods are rejected here because they limit the ability of the resulting system to function in a fully distributed manner. For any DDN, consider the situation where a cell A produces output items and places them on a data path which leads to cell B. The storage associated with the data path is considered to be local to B. Only cell A has the access permission to place data items in that storage when space is available. If space is not available then cell A must wait to complete its firing until space is freed in B's domain to hold the incoming data item. Implicit in this scheme is the existence of request-acknowledge communication protocols between cells, or between cells and data path FIFO stores. The placement of an output item corresponds to the movement of a piece of information from the local environment of A to the local environment of B. This locality of information guarantees that programs written as DDN's are side effect free. It also implies that storage can be physically distributed rather than central or shared. This is nice from both a cost and a performance standpoint for semiconductor storage mechanisms, which unlike disk and core find no advantage when

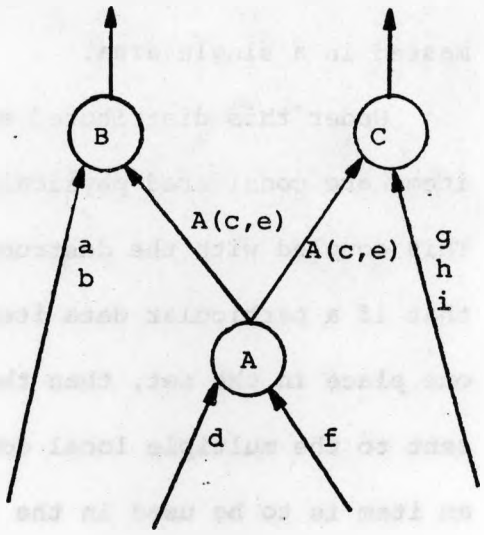
massed in a single area.

Under this distributed storage implementation the individual data items are considered physical storage entities as well as logical entities. This coupled with the destruction of data items in a firing set implies that if a particular data item is to be used concurrently in more than one place in the net, then that data item must be explicitly copied and sent to the multiple local domains where it will be used. Similarly if an item is to be used in the same place in a net but at several different instances of time (as would be the case in DDN iterative structures), then that data item must be copied or regenerated at each instance of usage and fed back into the net for successive use. The cases where these copies must be made depend upon the algorithm under construction. The mechanism for providing these copies could be a special primitive or merely the ability for any cell output to be copied an arbitrary number of times. In DDN's the latter approach was taken. Linguistically the choice is irrelevant except as a matter of style. Operationally however the special primitive suffers the disadvantage of greatly increasing the number of primitive operations that must be executed in a particular program. The advantage is that the sites where this copying activity takes place could also be distributed. For DDN's the disadvantage was felt to be too great (Dennis [4] has taken the other approach).

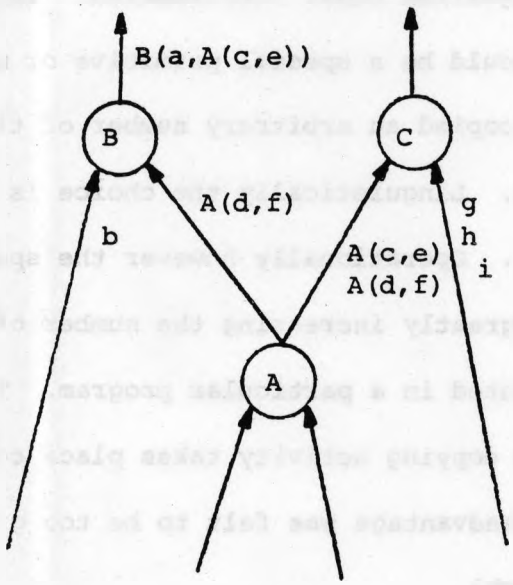
Figure 1 shows a simple abstract DDN program and snapshots of its execution. Small letters denote data items and capital letters denote cell functions. The firing set for cells A, B, and C is the set of all input data paths, and all cells have a single output, which is copied and placed on all output paths (cell A's output is copied and sent for possible concurrent use to B and C).



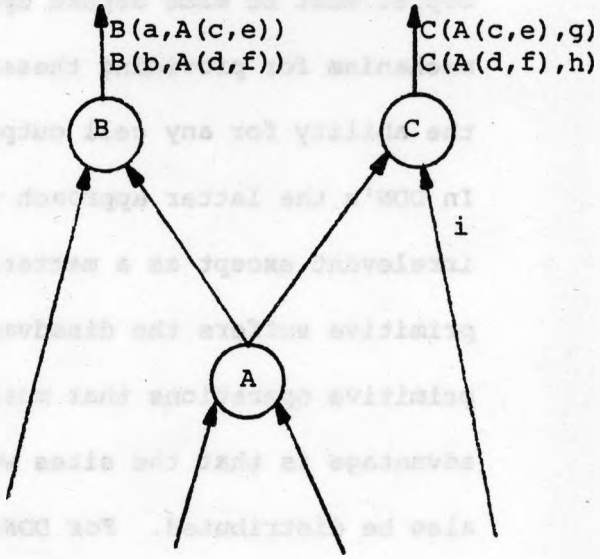
time t_0



time t_1: $t_1 > t_0$



time t_2: $t_2 > t_1$



time t_3: $t_3 > t_2$

Figure 1: Sample Abstract DDN Program

This simple example illustrates the asynchronous and distributed nature of the schema, as well as the inherent possibilities for two types of

concurrency. B and C are concurrent activities in that they are independent operations but at time t_1 A, B, and C are all fireable. A can be fired concurrently with B and C because of the capability for pipelined operation which is made possible by the FIFO data paths.

The firing rule for DDN cells implies that the snapshot history of Figure 1 is not unique. For example, between time t_1 and t_2 C could have fired but didn't. DDN's could be considered non-deterministic because no unique execution history can be determined. C1 and C2 actually imply that this is a possibility in fully distributed systems. However the following two properties do hold for DDN's:

DP1: Data items are persistent. Once placed on a data path items remain there until destroyed as a consequence of being in the firing set of a firing cell.

DP2: The FIFO organization of the data paths guarantees that the order of the items in any path cannot change.

DP1 and DP2 are necessary but not sufficient conditions to guarantee that DDN's are deterministic in the sense that they are output functional, i.e. that the order and values of the output data items of a DDN is deterministic. A DDN containing a cell with the function, "pick a random input item and put it on a random output path", would not be output functional. The cell types for DDN's have been chosen so that it is possible to determine by topological examination whether a given DDN is output functional or not.

To facilitate a substitution rule a DDN process (DDP) is defined to start with a single initial cell and terminate in a single final cell. This facilitates definitions about active and inactive processes as well as topological determination of certain net properties. As in any programming

language, it is possible to define illegal or meaningless DDP's. In analysis similar to compile type error analysis, the following topological statements about DDP's can be made:

1) whether or not the DDP may hang (i.e. is it possible for no cell to be fireable before the DDP outputs have been produced).
2) whether or not the DDP is safe for output functional operation in pipelined situations.

Facilities for dealing with error situations are also provided by the use of a special valued data item. Such correctness analysis is of increased importance for any asynchronous schema since there is no guarantee that subsequent "debug" executions will proceed in exactly the same manner. Such analysis is facilitated by the side effect free nature of DDN's and because influence is transparent in net programs.

The internal representation of DDN's in DDM1 is a variable length character string. Elements of the character string are also variable length character strings. The basic structure of this representation is described in the following BNF productions.

```
<DDN>::=(<NET NAME>(<CELL LIST>))
<CELL LIST>::=<CELL>|<CELL><CELL LIST>
<CELL>::=(<CELL TYPE>(<INPUT SLOT><OUTPUT LIST>))
<INPUT SLOT>::=(<INPUT MAP>(<INPUT LIST>))
<INPUT LIST>::=<INPUT>|<INPUT><INPUT LIST>
<OUTPUT LIST>::=<OUTPUT>|<OUTPUT><OUTPUT LIST>
<OUTPUT>::=(<DESTINATION LIST>)
<DESTINATION LIST>::=<DESTINATION>|<DESTINATION><DESTINATION LIST>
<DESTINATION>::=(<CELL LOCATION>)
```

where <CELL LOCATION>, <INPUT>, <INPUT MAP>, <NET NAME>, and <CELL TYPE> are all variable length, well nested, parenthesized strings containing the appropriate information. Furthermore <INPUT MAP> is a character string of length equal to the number of <INPUT>'s in the <INPUT LIST>. Each

character indicates the presence status of its associated input. The advantage of this variable length field structure is that it allows for convenient representation of nets of varying sizes. It also conveniently describes cells of arbitrary numbers of inputs and outputs, and any number of multiply copied outputs. This variable length character string is then interpreted during execution as described in section IV by the hardware of DDM1 in a character serial fashion.

The only sequencing rule for DDN programs is that of data dependency. A consumer of a data item must wait for the data item to be produced. This sequencing rule is in some sense the weakest possible and therefore DDN's are inherently highly concurrent descriptions of an algorithm. The concurrency is further enhanced by the ability of DDN's to be pipelined. This, combined with the asynchronous local autonomy of individual cells, makes it a simple task to decompose a DDN and distribute the smaller pieces (one or more cells) to sets of distributed physical resources. As such, DDN's serve as an excellent low level (it is not intended that anyone should program directly in DDN's) representation for distributed machine systems.

III. The Architecture

The architecture consists of sets of asynchronously communicating modules. The communication discipline is the standard four phase request-acknowledge protocol, which takes place on a two wire control link. The control link is used to sequence the activity on its associated data bus. This situation is illustrated and interpreted in Figure 2. A complete four cycle exchange takes place on a per character basis.

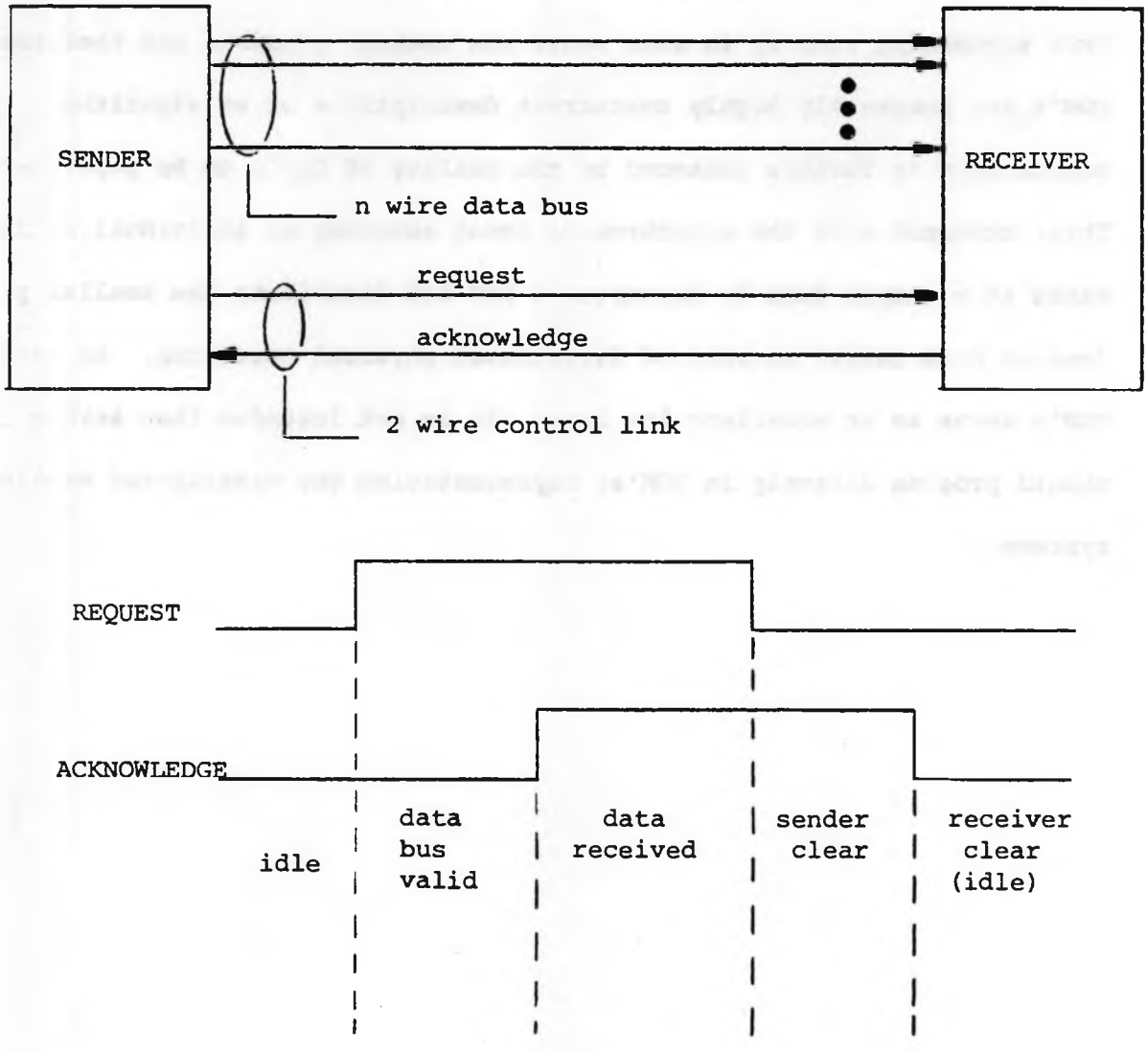


Figure 2: Asynchronous communication path and protocol interpretation

This protocol is not strictly speed independent in that it assumes that the relative propagation delays of the bus and the link lie within certain limits. A truly speed independent signaling scheme (Unger [10]) requires considerably more logic to implement. Typically the bus and link delays are comparable and the above protocol is considered to be sufficient for this architecture.

The basic computational unit of the architecture is a processor-store element (PSE). A PSE consists of a processor module P and its associated local storage module S. Any PSE can execute any machine language program providing that it has a sufficient amount of storage. No module that is not a PSE can perform this function. The architecture is a recursively organized set of these PSE's. The recursive definition of the structure is:

$\langle PSE_n \rangle ::= \langle P_n \rangle \langle S_n \rangle$

$\langle S_n \rangle ::= \langle ASU_n \rangle$

$\langle P_n \rangle ::= \langle AP_n \rangle \mid \langle AP_n \rangle \langle PSE\ GROUP_{n+1} \rangle$

$\langle PSE\ GROUP_{n+1} \rangle ::= \langle PSE_{n+1} \rangle \mid \langle PSE_{n+1} \rangle \langle PSE\ GROUP_{n+1} \rangle$

where subscripts denote the recursive level at which the module physically resides. <AP> is an atomic processor module, which has no further substructure (contains no PSE's). Similarly an atomic storage unit <ASU> has no PSE substructure. The width of a <PSE GROUP> has a physical bound. For DDM1 this bound is eight. The structure is depicted in Figure 3.

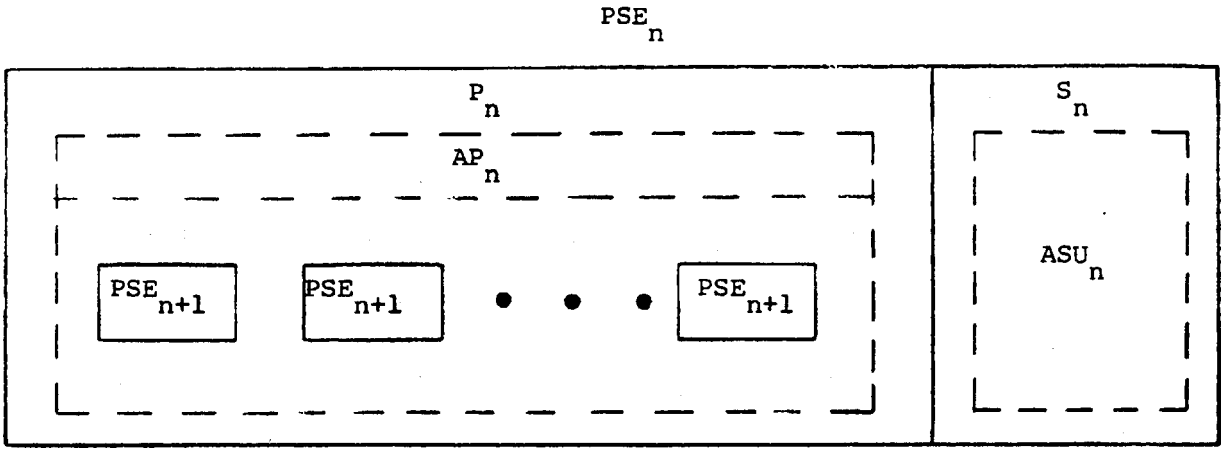


Figure 3: Recursive definition of PSE at level n

This structure allows a convenient notion for hierarchical distributed storage organization. Any S or ASU may consist of an arbitrary amount of storage of any desired medium. It will be seen that higher levels of PSE's are considered logically superior to lower level PSE's. As such, it is advantageous if at higher levels the S's (ASU's) are slower and larger than the S's (ASU's) of lower levels. The interface and functional ability of any ASU (regardless of size, speed, and level) is the same. The structure also allows for an arbitrary amount of P's that can be used concurrently. It is important to note that all AP's are identical regardless of level. However, the P's at higher levels will be more powerful (contain more substructure) than the P's at lower levels. More substructure implies more internal concurrent processing capability.

When viewed non-recursively this structure is simply a tree structure with a single root and a possibility for eight sons at any node. Each node of the tree is a PSE and capable of executing any machine language program. The leaf nodes have no substructure and therefore consist of an AP and an ASU. At each node the fan out is fixed but the depth of the tree is arbitrary. In this manner the architecture allows any desired

number of PSE's to be configured for a given machine. The desired goal is for machine performance to improve with the addition of more PSE's.

There are a number of ways to enforce this logical tree structure onto a collection of PSE's. The expense of crossbar switches vary as the square of the connected elements. For this architecture complete connectivity is not needed and also crossbars are not nicely extendible. Bus structures are also not indefinitely extendible in a convenient manner. Bus conflict would also drastically reduce actual parallelism in the machine. A goal of this architecture is to be able to support as much parallelism as PSE resource availability and the program permit. Therefore in DDM1 a simple 1 to 8 switch was chosen as the interface unit between successive levels of PSE's. The result is that the physical and logical recursive structure are the same. The structure is also fixed and cannot be dynamically changed. Upward traveling messages are passed on by the switch in an arbiter like manner. Downward going messages contain header fields which indicate their destination. This header is deleted by the switch as the message is passed. Downward and upward messages are dealt with by independent hardware, and therefore are controlled concurrently.

In keeping with the style for the internal representation of the machine language, information is passed between PSE's as messages which are variable length character strings. This character serial nature of the machine has the following advantages:

1) Hardware modules are made simpler and more applicable for LSI implementation due to the reduced pin count.
2) Hardware communication paths are more general in that variable length information units can be transmitted as varying numbers

of fixed width base characters. This facilitates a hardware substitution strategy for modules. Each module can interpret the variable length message and perform the indicated function. These advantages aid in greatly reducing the cost of the hardware modules. Some low level performance is lost by doing everything serially. The philosophy is to regain that lost performance many times over by providing a systems organization that allows for highly concurrent levels of activity.

Since DDM1 is merely a research prototype machine, a very simple 4 bit character set was chosen as the base alphabet. This 16 character alphabet includes the 10 digits, "(", ")", ".", ",", "-", and "MARK". "MARK" is a special character whose semantics change with the context. Similarly a simple arithmetic scheme was chosen. DDM1 does only integer arithmetic on sign magnitude, variable length integers.

Physical queues are placed between levels of PSE's in order to facilitate pipelining and increase physical module independence. Without queues the sender of a message would need to wait until the receiver could take the message. With queues, the message may be placed in the queue and both receiver and sender are free to proceed with their respective activities concurrently. If a queue becomes full, then the sender must wait until the receiver has freed up enough queue space. The new message can then be deposited and the sender would be able to resume further processing. If queue sizes are adjusted so that a sender only rarely is required to wait for space then the system would be well tuned for efficient processing. Optimal queue size depends on the process and the type of data items. It is therefore impossible to guarantee that no waiting will occur. The important thing is to insure that the system doesn't deadlock. This is insured in DDM1 by the strict hierarchical control and the restricted process structure.

A block diagram of the PSE structure is shown in Figure 4.

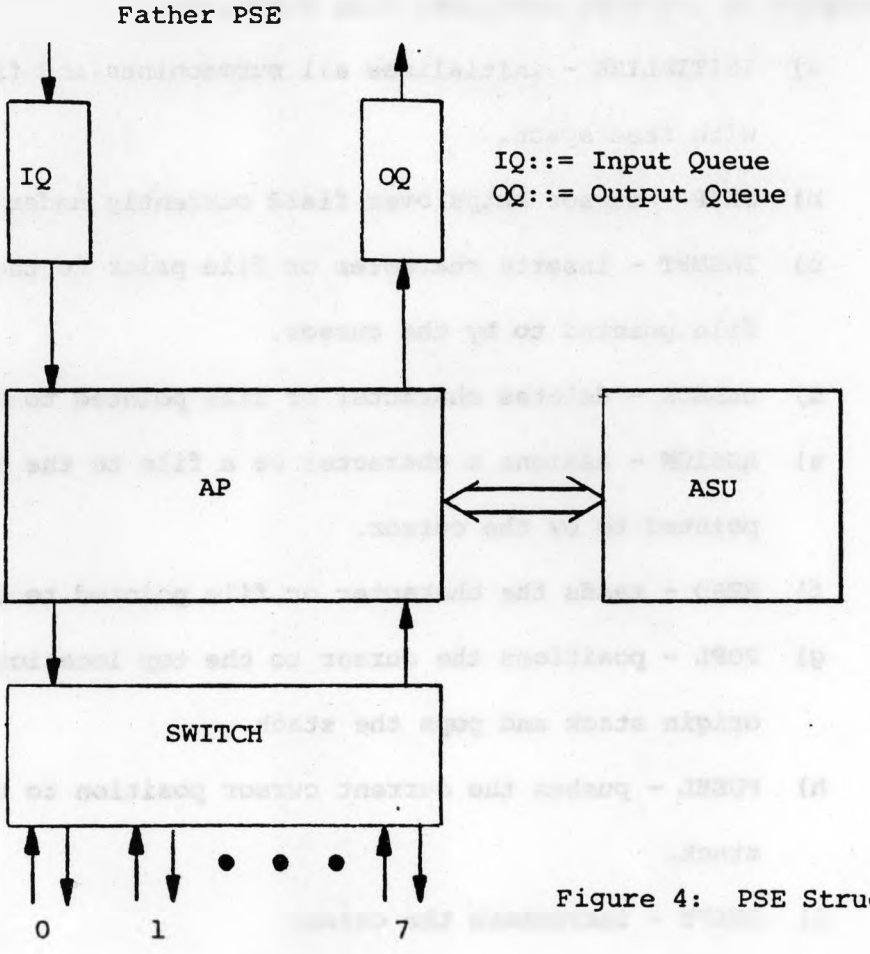


Figure 4: PSE Structure

All paths, except for the path between the ASU and the AP, are 6 wire paths (a 2 wire request-acknowledge control link and the 4 wire, character-width data bus).

The variable length, character serial message structure and net representation indicates that the ASU should be a highly flexible storage structure. Further requirements are that the ASU deal with pipelining of data items and their continual destruction upon cell firings. In order to increase the efficiency of the PSE, all of the storage management functions are performed internally by the ASU. The ASU appears as a

variable field length file system, which directly executes the following commands on its tree organized file structure:

a) INITIALIZE - initializes all submachines and fills the store with free space.

b) SKIP - cursor skips over field currently under the cursor.

c) INSERT - inserts character or file prior to the character or file pointed to by the cursor.

d) DELETE - deletes character or file pointed to by the cursor.

e) ASSIGN - assigns a character or a file to the character or file pointed to by the cursor.

f) READ - reads the character or file pointed to by the cursor.

g) POPL - positions the cursor to the top location of the local origin stack and pops the stack.

h) PUSHL - pushes the current cursor position to the local origin stack.

i) SHIFT - increments the cursor.

j) AINDEX - (absolute index) accepts node index vector and positions the cursor, starting at the first character in the ASU.

K) RINDEX - (relative index) does similar indexing operations starting from current cursor position.

The free space is managed automatically on all commands, as required, by the ASU hardware.

The ASU of DDM1 is a 4k 4 bit character store using random access static storage devices. The ASU is organized so that dynamic storage media can be easily accommodated. RAM store was chosen to minimize the

number of variables affecting performance measures. A black box view of the ASU is shown in Figure 5.

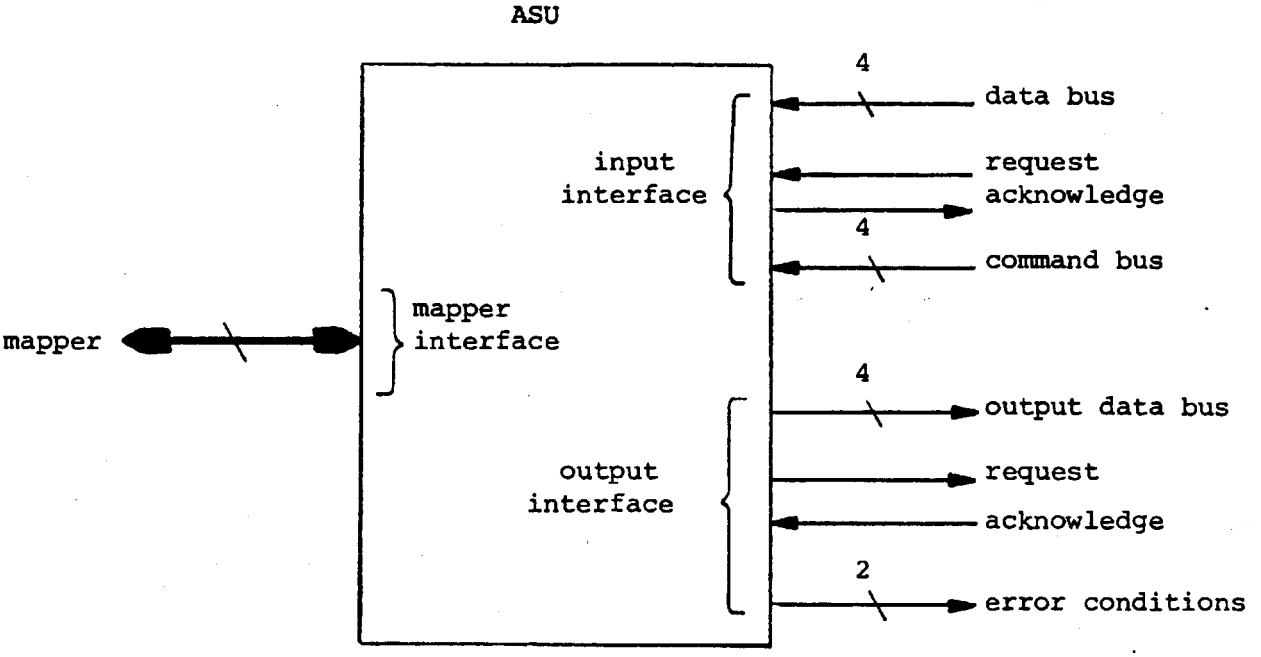


Figure 5: ASU Interface Structure

The error condition lines continually indicate whether an unrecoverable error has occurred or not. The state of these lines indicate the following conditions:

1) NO ERROR

2) FIXED SPACE AREA EXCEEDED (OVERFLOW)

3) ILLEGAL INDEX VECTOR

4) ERROR (ALL OTHER ERROR TYPES)

The mapper is an indexing speed up device and is an optional attachment to the ASU.

The internal structure of the ASU is shown in Figure 6.

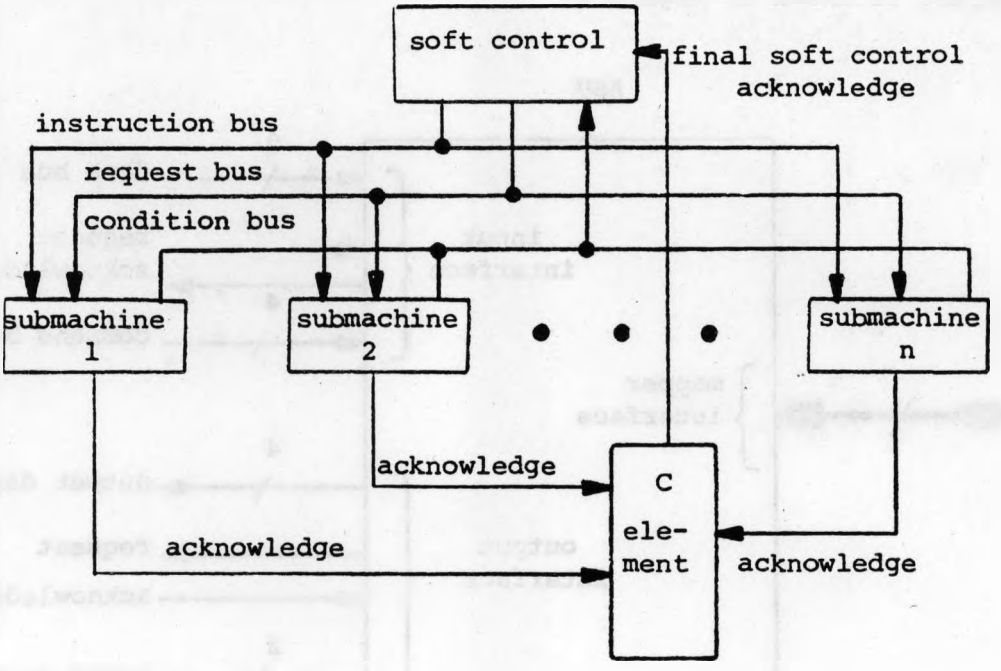


Figure 6. ASU Internal Structure

The soft control consists of a self-timed microprocessor-like control unit, a read/write microcode store, and a condition-select unit. The condition-select affects the contents of a condition register which can then be manipulated under program control. The soft control sequences operations for the submachines and also controls flow through the microcode. Submachines can be activated concurrently and all submachine instructions are controlled asynchronously. Submachines perform such functions as local-origin stack control, buffering for freespace, counting for index purposes, read/write control, etc. This structure provides a convenient mechanism for experimentation in that algorithms for storage management are easily changed in the microcode. Actual ASU hardware is easily changed by addition or removal of the asynchronous submachines

and changing some instruction decode ROM's.

All target locations in the ASU are found by a node index vector into the tree organized file structure. Such an ordered file structure is possible because of the choice of internal representation for the resident DDN programs. The use of index vectors and the dynamic nature of the ASU make absolute addresses useless. Such a mechanism requires large amounts of scanning to be done during indexing. There is a high probability of accessing certain nodes (i.e. those corresponding to a DDN cell) in the file structure. The mapper dynamically maintains valid absolute addresses for ASU file nodes down to a certain depth in the file tree. The map depth is selectable but currently not under program control. The 18 line mapper interface indicates ASU state information to the mapper. The mapper also monitors many of the ASU to AP lines. The storage of absolute ASU locations in the MAP corresponds to the preorder structure of the file tree. Preliminary statistical measurements on DDM1 indicate that the mapper speeds up indexing operations by a factor of 12.

The AP is intended to execute DDN programs composed from cells of 7 basic cell functions: synchronize, arbiter, operator, iteration control, call control, distribute, and select. The operator cell function is further subdivided into a set of operator codes. These are:

MONADIC

1) NEGATE

2) ABSOLUTE VALUE

3) BOOLEAN NOT

4) TYPE OF FILE (SUBSTRUCTURE OR NOT)

5) SIZE (COUNT NUMBER OF ITEMS IN A FILE)

6) REMOVE OUTSIDE PARENS ON MESSAGE

7) ADD OUTSIDE PARENTS TO MESSAGE

DYADIC

8) ADD

9) SUBTRACT

10) MAXIMUM

11) MINIMUM

12) MULTIPLY by 10^N

13) CATENATE

14) SPLIT AT N^{th} ITEM

15) INDEXED READ

16) EQUAL

17) NOT EQUAL

18) LESS THAN

19) LESS THAN OR EQUAL

20) GREATER THAN

21) GREATER THAN OR EQUAL

TRIADIC

22) INDEXED WRITE

The basic internal structure of the AP is the same flexible structure chosen for the ASU. In DDM1 the AP contains twice as many submachines as the ASU. The AP submachines perform such functions as arithmetic and relational operators, balance counting on message and ASU structures, an internal queue control called the AGENDA QUEUE (its use will be described in the next section), operand buffering, index counting, etc.

A more detailed description of the hardware of DDM1 can be found in [11] and [12].

IV. Modus Operandi

When a message corresponding to a DDN program enters a PSE at any level, the PSE may take one of two actions:

1) DECOMPOSITION AND ALLOCATION: if the PSE has substructure and if there exists some set of concurrent subnets in the DDN process, then the PSE may split the DDN and send the concurrent subnets to the next lower level PSE's.

2) EXECUTION: if the PSE has no subresources, or if there is no exploitable concurrency in the DDN, then the PSE executes the DDN at that level.

In addition to decomposition of the DDN, automatic resource allocation is done by placing concurrent subprograms in available physical resources. To aid the decomposition process, a structural description header may precede the incoming DDN in the message structure. This additional storage can greatly reduce time required for decomposition decisions in the PSE. In addition, each PSE must contain information about the number of available PSE's and the sizes of their respective stores. Problems would result if a DDN were sent to a PSE that was too large to fit in its local store. Only the local store size of immediate subresources is known. This insures the recursive nature of the decomposition process.

The decomposition process takes some time. It is important that the speed up gained by extra concurrency resulting from the decomposition is not overshadowed by the time to decompose. Experiments have indicated that a "first fit" decomposition is almost always better than a "best fit" decomposition strategy.

When a DDN is to be executed by a PSE, the DDN to be executed arrives as a message via IQ and is loaded by AP into ASU. The data items which cause cells of the net to become fireable also arrive as messages via the IQ. When a data item arrives, the message contains two subfields; 1) the destination, and 2) the value. The processor interprets the destination which is a DDN cell, and finds that cell in the ASU. If the incoming data item does not make the receiving cell fireable, then the item is stored in the appropriate place in the receiving cell. If the incoming data item makes the cell fireable then the processor fires the cell immediately, using the data item as needed. This scheme removes the need to place a field in store that is only going to be immediately read out and destroyed. When the processor produces a result, it formats a separate message of the form, (<DESTINATION><VALUE>), for each copy of the result that is needed. If the receiving cell for the message is not in the local store then the message is an external message and is placed in the OQ or SWITCH. If the receiving cell is in the local store then the message is an internal message and is sent to the agenda queue (AQ). After the AP has delivered all the result messages, it will accept messages from the SWITCH, the AQ, and the IQ (in descending order of priority. All AQ, IQ, and SWITCH result messages have the same format, and are processed in exactly the same manner as above. The priority scheme implies that as much work as possible is done in the low level fast areas of the machine before any new work is accepted from above. The only exception is the master clear signal which resets and initializes everything.

The processing algorithm can simply be viewed at the higher level as the processing of coroutines. One coroutine is the incoming message and the other coroutine resides in the ASU.

A number of other types of messages are also interpreted. The only one of importance here is the message used to borrow storage space from a parent. There are unexpected situations which can cause a PSE's local store to overflow. When this happens the PSE can logically borrow storage from a parent resource. This process may be repeated recursively. If the highest level PSE runs out of store then nothing more can be done. At this point the DDN under execution requires more storage than the system can provide and therefore cannot be executed. When storage is borrowed, performance of the system degrades drastically. It is therefore important to have a resource allocation strategy which minimizes such instances.

V. Conclusions

An architecture and systems framework for a fully distributed machine system has been presented which allows an unlimited number of processing elements to be combined in an asynchronous, recursive structure. The system is arbitrarily extensible and requires no special programming or hardware tuning to support extensions of any size. The system is capable of doing dynamic automatic resource allocation and can support very high levels of concurrent activity and pipelining. Regular use of a very few number of generic module types and the character serial nature of the processing scheme make system elements attractive for LSI implementation. The redundancy of processing elements makes a fail soft system possible.

Disadvantages of the system are:

1) The hard-wired, fixed-tree structure prohibits reallocation of unused PSE's to other branches of the architecture where these PSE's might be needed. This limits resource utilization in certain instances.
2) Redundant storage is required to facilitate decomposition.
3) The architecture doesn't take full advantage of the pipelining possibilities of the machine language.

DDM1, a version of the PSE structure, has actually been implemented and successfully executes programs in the DDN machine language. The work reported here on the function and structure of the ASU was based on some earlier work in this area by R. S. Barton, and G. Hodgman of Burroughs Corporation.

References

1. Davis, A.. SPL - A Structured Programming Language, Ph.D. Thesis, University of Utah (1972).

2. Davis, A. Data-Driven Nets - A class of maximally parallel, output-functional program schemata. Burroughs IRC Report, San Diego (1974).

3. Davis, A. Data Driven Nets - A maximally concurrent, parallel process representation. Burroughs ASDO Report, San Diego (1977).

4. Dennis, J. B.. First version of a data flow procedure language. Lecture Notes in Computer Science, 19, Springer-Verlag, New York (1974), 362-376.

5. Rodriguez, J. E.. A Graph Model for Parallel Computation. Technical Report MAC TR-64, Laboratory for Computer Science, M.I.T. (1969).

6. Glushkov, V. M., et. al. Recursive Machines and Computing Technology. IFIPS Proceedings 1974, North Holland, New York.

7. Miller, R. E., and J. Cocke. Configurable Computers: a new class of general purpose machines. Report RC 3897, IBM, New York (1972).

8. Arvind, and K. P. Gostelow. A computer capable of exchanging processors for time. Information Processing '77, North Holland, New York (1977).

9. Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM 1974 National Conference, (1974), 402-409.

10. Unger, S.. Asunchronous Sequential Switching Circuits, Wiley & Sons, New York (1969), 243-252.

11. Davis, A. L.. An overview of data-driven machine #1. Burroughs ASDO Report, San Diego (1976).

12. Barton, R. S., A. L. Davis, et. al. System and method for concurrent and pipeline processing employing a data driven network. U. S. Patent No. 3,978,452, issued August 31, 1976.